APPARATUS AND METHOD FOR SKIPPING SONGS WITHOUT DELAY

BACKGROUND OF THE INVENTION

TECHNICAL FIELD

This invention generally relates to Internet based personalized radio/music technology. More particularly, the invention relates to an apparatus and method allowing a user to skip one or more songs in a pre-selected play list without having an unintended delay between skips.

DESCRIPTION OF THE RELATED ART

Internet based personalized radio services, such as Radio@AOL and iSelect, provide users a high flexibility to choose programs and make their own play list using a graphical user interface which is part of the client application of the service. The client application sends a user's request to the server and the server responds to the user's request by returning the requested in compressed data. The client application along with the user's browser executes a decompression algorithm to decompress the compressed data in real time and sequentially plays the data as it is transferred from the server to the user's computer over the Internet. Using streaming technologies, the user's computer does not need to download the entire file first and then play it. Rather, after downloading a minimal section of data into a buffer, the user's computer reads from the buffer and plays the song or music represented by the data. The data already read by the computer is deleted from the buffer so as to ease the RAM requirements and maintain a balance between the write-in and the read-out data flows.

When the user's computer plays a play list or a preset, which is either created by the user or by the service provider, the server sends and the user's computer receives the data over the Internet in a programmed sequence so that there is no unintended delay between any two programs in the list. If the user does not interrupt, the computer plays the songs in the list one by one in an organized consecutive manner. However, when the user switches from one list or preset to another, the users actually

interrupts the natural flow of the play list or the preset. In these circumstances, because the computer has to request that the server start to send the data for the target list or preset, several seconds of loading time is needed. Likewise, when the user wants to be actively be involved in the sequence of the play list or preset by skipping one or more songs, as it is illustrated in FIG. 2C, the natural flow of the pre-determined play list or preset is interrupted by the loading transition. This type of unintended delay between skips has been a major factor affecting users experience using personalized radio service.

Therefore, there is a need in the art to provide a solution to overcome the unintended delay or pause problem caused by a user's skipping from one song to the other while a pre-determined list of selections is playing in a programmed sequence.

SUMMARY OF THE INVENTION

In an Internet based personalized radio, where a user has a pre-selected list of songs to be played in a particular order, the invention provides an apparatus and method allowing the user to skip one or more songs without having a delay between skips. This is accomplished by downloading and pre-caching, i.e. pre-buffering the first small portion of each of the next several songs on the play list so that, should the user choose to skip to any of the next several songs, the pre-buffered small portion of the target song is already available to be played and therefore there is no unintended delay between two songs. The apparatus starts to play the pre-buffered small portion of the target song and starts to download the rest of the target song at the same time. Because the system is so configured that the time for playing the pre-buffered small portion is longer than the initial buffering time for the rest of the target song, the entire target song is played smoothly. In other words, there is no unintended delay between the first small portion and the rest portion either.

In the preferred embodiment of the invention, the first small portion is approximately the first ten seconds of the song. This solution is advantageous because ten seconds of pre-buffering complies with various royalty requirements such that if the user skips before the ten seconds pre-buffered portion is played, a royalty is not accessed for listening to the song. In addition, avoiding of downloading the entire next song conserves bandwidth and memory.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1A is a schematic diagram illustrating a system in which a user uses a graphical interface running on a local computer to access a radio service provided by a remote server over the Internet;

FIG. 1B is a block diagram further illustrating the local environment in which the preferred embodiment of this invention operates;

FIG. 2A is a schematic diagram illustrating a natural flow of the user's play list, which is a sequence of songs pre-selected by the user via the graphical user interface supported by the client application in FIG. 1A and FIG. 1B;

FIG. 2B is a schematic diagram illustrating how a buffer works;

FIG. 2C is a schematic diagram illustrating how the natural flow of the user's play list is interrupted by skips;

FIG. 3A is a schematic diagram illustrating a pre-buffering solution according to the preferred embodiment of this invention;

FIG. 3B is a flow chart illustrating a process according to the pre-buffering solution according to FIG. 3A;

FIG. 3C is a flow chart further illustrating a major loop of FIG. 3B; and

FIG. 3D is a flow chart further illustrating another major loop of FIG. 3B; and

FIG. 4 is a schematic diagram illustrating the data capacity of a communications channel being shared by the streaming for playing a song and the streaming for downloading.

DETAILED DESCRIPTION OF THE INVENTION

Referring to the drawings, in particular to FIG. 1A and Fig. 1B, which, in combination, illustrates an environment where this invention embodies. FIG. 1A is a schematic diagram illustrating a system in which a user uses a graphical interface110 running on a local computer 120 to access a radio service provided by a remote server 130 over the Internet 140. The local computer is powerful enough to execute in real time a decompression algorithm required for streaming.

FIG. 1B is a block diagram further illustrating the local environment in which the preferred embodiment of this invention operates. The local environment includes a

computer platform 121 which includes a hardware unit 122 and an operating system 123. The hardware unit 122 includes at least one central processing unit (CPU) 124, a read only random access memory (usually called ROM) 125 for storing application programs, a write/read random access memory (usually called RAM) 126 available for the application programs' operations, and an input/output (IO) interface 127. Various peripheral components are connected to the computer platform 121, such as a data storage device 128, an audio system 129 such as an MP3 player, and an Internet connection interface such as an Ethernet 107. The user uses a web browser 108 to go online. An Internet radio client application 109, which supports the graphical user interface 110, runs on the computer platform 121. The client application 109, along with an advanced clip loader which is also called piledriver, can be deployed as a plug-in to the web browser 108 such as the Netscape Navigator. Those skilled in the art will readily understand that the invention may be implemented within other systems without fundamental changes.

Using clip based data streaming technologies, the user's local computer 120 can play audio or video program in real time as it is being downloaded over the Internet as opposed to pre-storing the entire program in a local file. The Internet radio client application 109 coupled to the web browser 108 decompresses and plays the data as it is being transferred to the local computer 120 over the Internet. The piledriver is responsible for delivering, for example, an Ultravox formatted stream to the client application in a seamless fashion, in addition to raw data. Streaming audio or video avoids the unintended delay entailed in downloading an entire file and then playing it with a helper application. For the clip based streaming to work, the client side receiving the data must be able to collect the data and send it as a steady stream to the program that is processing the data and converting it to sound or pictures. This means that if the data does not come more quickly enough, the presentation of the data will not be smooth. If the streaming client receives the data more quickly than required, it needs to save the excess data in a buffer, which is an area of memory in the write/read random access memory (RAM). Even when the write speed and the read speed are exactly same, to maintain a smooth data flow, a minimum amount of data in the buffer is necessary.

From a high level view, the piledriver receives a play list from an audio or video client application. It analyzes the play list and locally caches the first small portion (e.g. first ten seconds) for clip in the play list. The client can then connect to the piledriver data pump and retrieve the data stream using the HTTP or Ultravox 2.0 Protocols. The major functions of the piledriver include: (1) managing the retrieval and caching the pre-butter for items in the play list; (2) managing the content in memory; (3) providing content to audio or video clients using raw data or the Ultravox 2.0 protocol from either a local cache or directly from a content-store; and (4) providing a stream of data to the audio or video client mimicking local disk functionality.

The piledriver takes a play list and attempts to present an uninterrupted stream of audio or video output to the client application. One of the primary features of the client application is that it allows the listener to abort a current song being played and request the start of the next clip in the play list.

In order to minimize the amount of time taken for skipping, the piledriver performs two operations in parallel. First, it requests the first URL in the play list from the UltraMODS/HTTP server. Once the pre-buffer data arrives, it waits for the audio or video client to start playing the clip, and also continues downloading the pre-buffer segments for each of the next clips in the play list, in order.

There are two reasons to request the pre-buffers in advance. First, it reduces the delay involved in requesting the clip and then obtaining the pre-buffer before being able to play the audio or video. Second, it causes UltraMODS/HTTP to obtain the media file from the content-store if it does not have it already, hopefully in advance of the new by the client.

The functional components for the piledriver include a pre-buffer cache engine and a clip/stream retrial application program interface (API). The pre-buffer cache engine is responsible for caching clips in advance of playtime. The clip/retrial API contacts the Apache/UltraMODS/Cache engine for content. For illustration purpose, given below is an exemplary list of API calls and their functions:

pdInit

PILEDRIVERTYPE *pdInit(int cacheahead, int initringsize)

This is the first function called to initialize the piledriver. The number of clips to cache in advance and the size of the pre-buffer cache can be specified.

pdAddItem

PDFILEHANDLE pdAddItem(PILEDRIVERTYPE *piledriver, char *url, unsigned long start, unsigned long end)

Call to add a URL to the cache-ahead playlist. It can be configured to add the entire play list, or just enough to keep the cache-ahead system busy.

pdOpen

PDFILEHANDLE pdOpen(PILEDRIVERTYPE*piledriver, PDFILEHANDLE handle)

Call to open PFFILEHANDLE after the item has been added to the cache engine with pdAddItem. If the file is cached it returns the size of the pre-buffer, 0 if no pre-buffer, or –1 if there was an error related to the files availability.

pdReadRaw

int pdReadRaw(PILEDRIVERTYPE *piledriver, char *buffer, unsigned int toread, PDFILEHANDLE handle)

Call to an opened PFFILEHANDLE to retrieve data. The size of the data is returned, 0 if none, -1 if EOF (End of File) has been reached or the connection was broken.

pdReadCooked

int pdReadCooked(PILEDRIVERTYPE *piledriver, char *buffer, int toread, unsigned short *msgtype, PDFILEHANDLE handle)

Call to an opened PFFILEHANDLE to retrieve Ultravox messages. The size of the data is returned, and msgtype contains the clad and type of the Ultravox message. 0 is returned if no message is available, and –1 if EOF has been reached or the connection was broken.

pdClose

int pdClose(PILEDRIVERTYPE *piledriver, PDFILEHANDLE handle)

Call to close and remove the cache-ahead engine a PFFILEHANDLE. Always call this function even if the file failed to open.

pdDeInit

int pdDeInit(PILEDRIVERTYPE *piledriver)

Call to stop all cach-ahead transactions, close and remove all open PFFILEHANDLEs and free all used memory.

Error Notification

Call to make error notification. In the event of an error in any of the API functions, PILEDRIVERTYPE->error and PILEDRIVERTYPE->error-buffer contain the error code and the error string associated with the current error condition. Error codes are located in PDRIVER.H

FIG. 2A is a schematic diagram illustrating a natural flow of the user's play list, which is a sequence of songs pre-selected by the user via the graphical user interface 110 supported by the client application 109 in FIG. 1A and FIG. 1B. Upon the user's command to play his play list, the client application 109 first checks whether there is a file characterized as the first song (S_1) of the play list is available in the buffer. If not, then start downloading the data from the server 130 over the Internet. After an initial buffering time 210, the sequence of songs is played in a continuous manner. FIG. 2B is a schematic diagram illustrating a buffer 211 which is an area of memory for temporarily storing the data downloaded from the server 130 over the Internet. The buffer 211 is used to decouple processes so that the reader 213 and writer 212 may operate at different speeds or on different sized blocks of data. For smooth playing a song or a sequence of songs, the initial buffering time 210 is necessary.

However, when the user chooses to skip to a next song before the current song ends, the natural flow is interrupted because it takes time to send the skip command to the server which starts to transmit the data for the next song, and thus a new period of buffering time is required before the next song starts to play. For example, as illustrated in FIG. 2C, after the initial buffering time 210, the first song S_1 starts playing at the time t_1. Before the first song S_1 ends, the user decides to skip to the second S_2 at the time t_a. When the application receives the command to skip to the second song S_2, the reader 213 stops reading and the writer 212 stops writing the rest data for S_1, and at the same time the application notifies the server to stop transmission of the data for S_1. Then, the application checks whether there is a file characterized as S_2 in the buffer 211. Because S_2 is not downloaded yet, the application sends the server a request to transmit the second song S_2. Thus, a buffering time 220 (from t_a to t_b) is needed before S-2 starts. After the buffering time 220 (from t_a to t_b), the reader 213 starts to read S_2 at the time t_b. Similarly, when the user decides to skip to the third song S_3 before the S_2 ends, a buffering time 230

(from t_c to t_d) is needed before S-3 starts at time t_d. Because each buffering time is about several seconds, the music flow 250 is interrupted and the user experience is affected.

FIG. 3A and FIG. 3B are schematic flow diagrams illustrating a solution to overcome the problems as illustrated in FIG. 2C. The solution includes the following steps to be executed by the computer:

Step 301: Start downloading the second song S_2 immediately after the initial buffering time 210 is over at the time t_1. This step is called pre-buffering or pre-caching. The application is configured to download only the first few seconds of S_2. In the preferred embodiment, the application is configured to download the first ten seconds. After download the first ten seconds of the second song, start downloading the first ten seconds of the third song. The similar pre-buffering step goes so on and so forth. In the preferred embodiment, the application is configured to download and pre-buffer the first ten seconds of five songs subsequent to the current song which is being played. The total time required for pre-buffering five songs is about one minute. Usually a user would be able to decide whether or not to continue the song after listening to it for one minute. Therefore, although the application can be otherwise configured, pre-buffering five songs would be good enough for most of circumstances.

Step 302: Assuming the user decides to skip to a target song (for example S_5), the application first check whether there is a file characterized as the target song.

Step 303: If S_5 is identified in the buffer and because the first ten seconds of S_5 is already there, the system can start to read S_5 immediately. This means that there is no unintended delay between S_1 and S_5 unless the networking condition is abnormally bad or the user has exhausted the local cache. At the same time, the application asks the server to transmit the rest of S_5 to the buffer. Because the buffering time for the rest of S_5 is less than ten seconds, by the time the reader finishes reading the pre-buffered ten seconds of S_5, a sufficient part of the rest of S_5 is already there and is ready to be read. Therefore, there is no interruption between the first ten seconds of S_5 and the rest of S_5. In this way, the user experience is enhanced and waiting time is minimized.

Step 304: While the song (S_5) is being displayed, update Step 301 to keep five songs subsequent to the current one being pre-buffered.

Step 305: Play next song after S_5 is over.

Step 306: Repeat Step 302 if the user wants to skip while S_5 is being played.

Steps 301-306 represents the first loop in which the user's play list is played without interruption even he sometimes decides to skip one or more songs.

This invention also helps to bring the song playing back into the first loop when an interruption occurs.

Step 310: If the check result in Step 302 is no (i.e. the target song S_5 is not identified in the buffer), the system requests that the server stop transmitting the prior song (S_1 in the example) and start transmitting the target song.

Step 311: Start to download the target song. Because the target song is not pre-buffered, an initial buffering time is required before the target song can be played. During initial buffering time, typically 5-6 seconds, the system is silent.

Step 312: Start to play the target song. This step leads to step 305 or step 306, and step 301. Because the system always attempts to have five next songs pre-buffered, if the target song is one of the pre-buffered, the natural flow of the play list will not be interrupted by skipping.

When the user skips to the target song, the pre-buffered songs which are prior to the target in the play list (e.g. S_2-S_4 if the user skipped from S_1 to S_5) will be deleted from the memory just as they had already been played.

If the application is configured to keep the skipped pre-buffered data for a short period of time, for example for 10 seconds, the user could, though not very much meaningful for many people, come back to any of the songs before it is deleted from the buffer.

FIG. 3C and FIG. D are flow charts further illustrating the various loops according to FIG. 3B.

Referring to FIG. 3C, step 330 actually includes following two sub-steps:

- as soon as a song starts to play, download, consecutively, a first small portion (e.g. ten seconds) of each of a number of songs which are, in the pre-determined sequence (i.e. play list), subsequent to the song which is currently playing; and

- pre-cache the downloaded small portions in a buffer which is an area of the user's computer memory.

In step 330A, assuming the user skips to a song (called target song) in the play list before the song in playing is over, the computer checks whether the target song belongs to one of these pre-cached in step 330 by checking whether a file characterized as the target song exists in the buffer. If yes, go on to step 330B in FIG. 3D.

Referring to FIG. 3D, step 330B includes the following sub-steps:

- play the first small portion of the target song;
- start to download the rest of the target song (by identifying a ten seconds mark, for example); and
- delete any pre-cached song which is prior to the target song in the pre-determined sequence.

Note that as soon as the pre-cached portion of the target song starts playing, step 330 needs to be updated. In particular, if one or more songs subsequent to the target song are already pre-cached, skip them and download the subsequent ones, executively, to make up the pre-designated number (five, for example).

In step 337, when the playing of the pre-cached portion ends, immediately play the rest of the target song which is being downloaded from the server over the Internet.

In steps 338-339, if the user does not want to skip to another song while the target song is playing, then play the next song in the sequence, and at the same time, delete any pre-cached song which is prior to this song. As soon as this song starts playing, step 330 needs to be updated. Because all pre-cached files, which are prior to this song in the sequence, have been deleted from the buffer, the user's computer must send request to the server to transmit the first small portion (e.g. ten seconds) of a designated number of songs, one by one. Then, the user's computer downloads and pre-caches these files in the buffer.

If the user wants to skip to another song before the playing of the target song in steps 330B-337, the process continues on step 330A in FIG. 3C which illustrates another loop.

No referring to FIG. 3C, in step 300A, the user's computer checks whether the new target song is already pre-cached by checking whether a file characterized as the

new target song exists in the buffer. If not, go to step 331 which includes two sub-steps:

- send request to the server to stop transmitting of the song in playing and to start transmitting the new target song; and
- at the same time, delete the pre-cached portion for any song which is prior to the new target song in the designated sequence of songs.

Then, start to download the new target song in step 332. Because the new target song is not pre-cached, it takes a short period of buffering time (about five seconds) before the computer can play the song. This buffering time causes the interruption of the natural flow of the user's play list. This invention helps minimize the occurrences of the interruption. If the user always skips to a pre-cached song, no interruption would occur at all unless the networking condition is abnormally bad or the user has exhausted the local cache.

In step 333, as soon as the buffer allows, play the new target song while it is being downloaded. At the same time, update step 300 by deleting outdated pre-cached files (i.e. the pre-cached portions of the songs prior to the new target song) and pre-buffering the subsequent songs. This step is important because it helps the user to return to the none-interruption loop.

Assuming the user does not to skip again while the new target song is playing, go to step 335 which includes the sub-steps of:

- as soon as the playing of the new target song ends, play the first small portion of the next song subsequent to the new target song;
- at the same time, download the rest of the "next song" (beginning from the ten seconds mark, for example); and
- update step 300, wherein if one or more songs subsequent to this "next song" are already pre-cached, skip them and download the subsequent ones, executively, to make up designated number.

Then, in step 336, play the rest of the "next song" as soon as the pre-cached portion ends.

If the user wants to skip again, the loop starting at step 300A will be repeated.

The pre-caching (i.e. the pre-buffering) solution described above is possible because the total capacity of the communication channel can be shared between several

independent data streams using some kind of multiplexing, in which, each stream's data rate may be limited to a fixed fraction of the total capacity. As it is illustrated in FIG. 4, the data transfer rate for a regular DSL communications channel ranges from 256K to 8M byte per second (bps), the voice conversations and music signals only use 64K bps. Therefore, the streaming track for the downloading data for pre-buffering can use the rest capacity.

The solution described above can also be used in Internet based video service and any other services where an initial buffering time is needed before the first section of the downloaded data can be read.

In the preferred embodiment of the invention, the first small portion is approximately the first ten seconds of the song. This solution is advantageous because ten seconds of pre-buffering complies with various royalty requirements such that if the user skips before the ten seconds pre-buffered portion is played, a royalty is not accessed for listening to the song. In addition, avoiding of downloading the entire next song conserves bandwidth and memory.

In view of the different possible embodiments to which the principle of this invention may be applied, it should be recognized that the preferred embodiment described herein with respect to the drawings is meant to the illustrative only and should not be taken as limiting the scope of the invention. One skilled in the art will readily appreciate that other applications may be substituted for those set forth herein without departing from the spirit and scope of the present invention.

Accordingly, the invention should only be limited by the Claims included below.